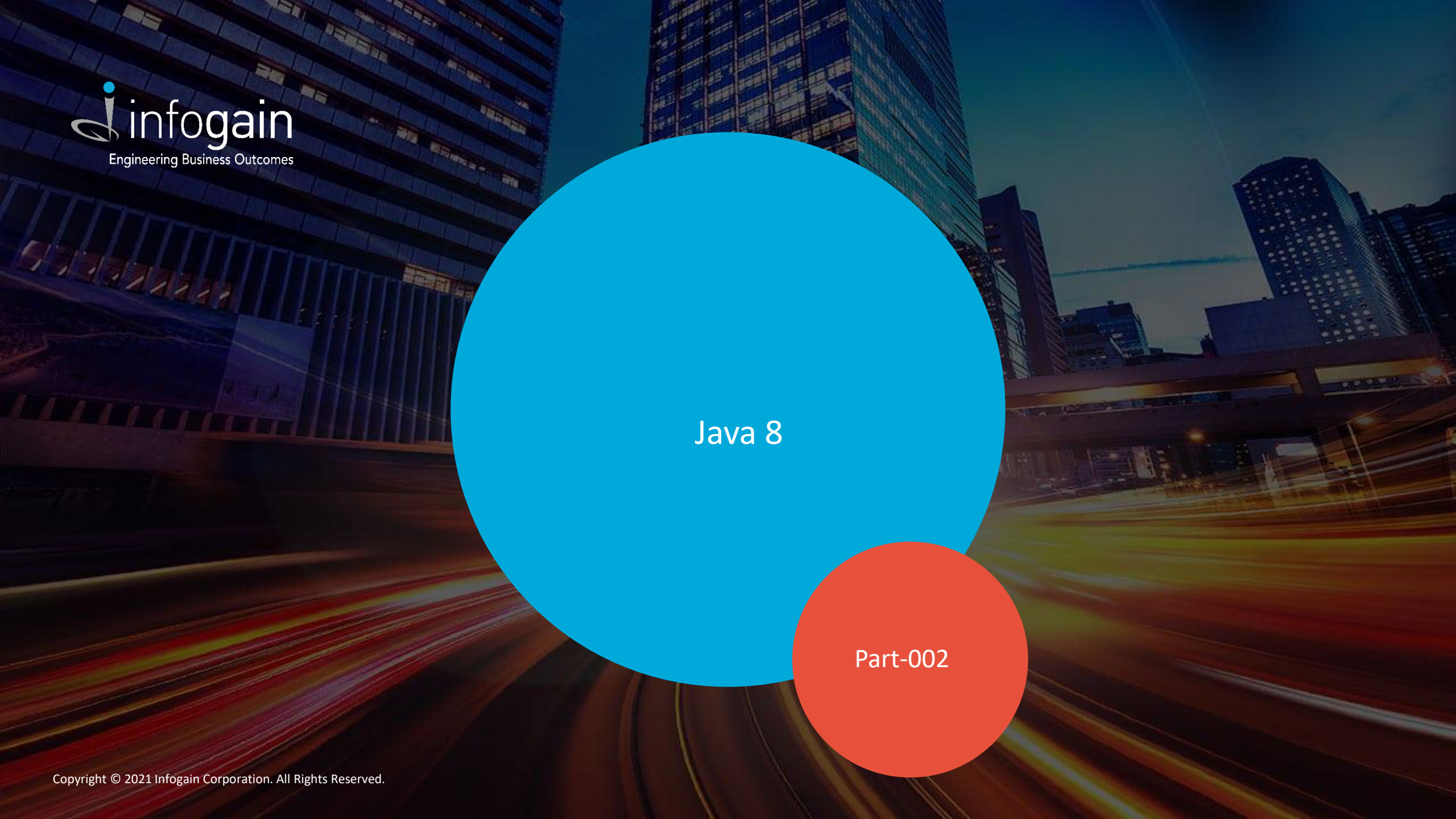

Objectives

- Able to explain and build Stream
- Able to implement Map ,Filter and Reduce operations
- Able to explain and use Stream Collectors
- Able to explain and use Optional class
- Able to use various methods of Optional class

What Is a Stream?

- **java.util.stream.Stream Interface**
  - Stream is a sequence of elements from a source. Source can be anything like **collections**, **arrays**, or **I/O** resources etc.
  - It provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
  - It supports operations like **filter, map, limit, reduce, find, match**, and so on.
  - It allows pipelining because most of the stream operations return stream itself so that their result can be pipelined.
  - It is designed for lambdas and do not support indexed access
- **What does it do?**
  - It provides a way to efficiently process large amounts of data
- **Why can't a Collection be a Stream?**
  - Because Stream is a new concept, and we don't want to change the way the Collection API works

Collections VS Streams

- Collections are mainly used to store and group the data In **ArrayList , LinkedList** etc .
- Streams are mainly used to perform operations on data.
- You can add or remove elements from collections. You can’t add or remove elements from streams.
- Streams are traversable only once. If you traverse the stream once, it is said to be consumed.
- To traverse it again, you have to get new stream from the source again. Collection can be traversed multiple times.

How can we build a Stream?

- **Example of building Stream from a Collection using the stream() method;**

```
Stream<Integer> stream = list.stream();
stream.forEach(p -> System.out.println(p));
```

- **Example of building strem Using Stream.of(T[]) Static Factory Method**

```
Stream<Integer> stream = Stream.of( new  Integer[]{1,2,3,4,5,6,7,8,9} );
stream.forEach(p -> System.out.println(p));
```

Stream Operations

- **Stream Operations**
  - Stream operations are broadly divided into **intermediate** and **terminal** operations that are combined to form pipeline of tasks.
  - A stream pipeline consists of a source **(such as a Collection, an array, or an I/O channel);** followed by zero or more intermediate operations such as **Stream.filter** or **Stream.map**; and a terminal operation such as **Stream.forEach** or **Stream.reduce**.
- **What is Intermediate operation?**
  - An intermediate operation returns stream back again. Which allows you to chain multiple method calls in a row.
  - Intermediate operation don't modify the original stream, everytime they return a new stream.
- **Example of Intermediate operation**
  - filter(),map()etc.

Stream Operations(contd..)

- **What is Terminal Operation?**
  - Terminal operation traverse the stream and execute the pipeline of intermediate operations to produce the result of certain type.
  - After the terminal operation is performed, the stream pipeline is considered as consumed, and can no longer be used.
  - A stream implementation may throw **IllegalStateException** if it detects that the stream is being reused.
- **Example Terminal Operation**
  - forEach() ,
  - collect() etc.

Stream API

- Provides various methods that let you to implement various operations such as **consuming**, **filtering, slicing, mapping, finding, matching and reducing.**
- **Consuming the Stream**
  - Stream provides **forEach** method which accepts a **Consumer** as argument and performs the action on each element.
  - void forEach(Consumer<? super T> action)
  - The forEach is the terminal operation and returns void.
- **Filtering Operation**
  - Stream interface provides a method **filter** which accepts a **Predicate** as argument and return a stream that matches the given predicate.
  - The predicate will be applied to each element to determine if it should be included to new stream.
  - you can pass lambda expression to this method as filtering logic.
- **Example**

```
memberNames.stream().filter((s) -> s.startsWith("A")) .forEach(System.out::println);
```

Stream API(contd..)

- **Mapping Operation**
  - Stream provides **map** method which accepts a **Function** as argument and returns a new stream consisting of the results of applying the given **function/transformation** to the elements of this stream.
  - It allows you to transform your data you can supply your transformation logic to map() method as lambda expression and it will transform each element of that collection and return a new stream.
- **Example**

```
memberNames.stream().filter((s) -> s.startsWith("A"))
        .map(String::toUpperCase)
        .forEach(System.out::println);
```

Stream API(contd..)

- **Reductions**
  - Reductions are terminal operation which refers to the process of combining all elements in the stream repeatedly to produce a single value which is returned as the result of the reduction operation.
  - It applies a binary operator to each element in the stream and trigger the processing of data.
  - Reduction operations are always present at the end of a chain of Stream methods.
- **Method Signature**
  - T reduce(T identity, BinaryOperator<T> accumulator)
- Here, identity is the initial value of type T and accumulator is a function for combining two values.

Stream API(contd..)

- **How Reduction works?**

```
List<Integer> numbers = Arrays.asList(20,30,50);
Integer sum = numbers.stream().reduce(0,(num1,num2) -> num1+num2);
System.out.println("Sum is " + sum);
```

- **1st argument :**
  - The identity is the initial value of the type T which will be used as the first value in the reduction operation
- **2nd argument:** Reduction operation of, type **BinaryOpertor<T>**
- **A BinaryOperator is a special case of Bifunction.**
  - It represents a binary operator which takes two operands and operates on them to produce a result
  - T: denotes the type of the input arguments and the return value of the operation

Identity Element

- **What happens if the Stream is empty ?**
  - The reduction of an empty Stream is the Identity element
- **What happens if the Stream has only one element?**
  - If the Stream has only one element, then the reduction is that element
- **What happens if the Stream has multiple elements?**
  - If the Stream has multiple elements, then the reduction is of all that elements

Stream API(contd..)

- **Built-In Reduction Method**
  - Stream API provides various built-in reduction to perform specific operation and return value of certain type.
- **Numeric Reduction**
  - min()
  - max()
  - Count()
  - Sum()
- **Boolean Reduction**
  - allMatch(), noMatch() , anyMatch()
- **Reductions that returns an Optional Instance**
  - findFirst() , findAny()

Stream API(contd..)

- **Matching**
  - Many-a-times you need to check whether object(s) in the given stream match a specific criteria. Instead of writing logic for iterating over the stream elements and checking each object whether it matches the criteria you can use methods provided by Stream API.
  - Stream API provides **anyMatch**, **allMatch** and **noneMatch** terminal operations which takes a **Predicate** as argument and returns a boolean result by applying the **Predicate** to the elements of the stream.
  - Predicate might not be applied to all the elements if further execution is not required.
- **anyMatch**:
  - Returns true if any element found matching with the predicate. Predicate will not be applied to other elements if any matching found.
- **allMatch:**
  - Returns true if all elements are matching to the given predicate.
- **noneMatch**:
  - Returns true if none of the elements are matching to the predicate.

Stream API(contd..)

- **Finding element**
  - **findFirst()**
    - Returns first element of a stream wrapped in an Optional object.
  - **findAny()**
    - Randomly returns any one element in a stream. The result of this operation is unpredictable. It may select any element in a stream.
- **Counting Element**
  - **count()**
    - Returns number of elements in stream as a long.
- **Example**

```
long total= memberNames.stream().filter((s) -> s.startsWith("A")) .count();
        System.out.println(total);
```

Collectors

- **Collector**
  - Collector is nothing but a mutable reduction operation that accumulates elements from the stream into a mutable container like **ArrayList** , **HashSet** and **Map** etc.
  - Stream interface contains a most frequently used terminal operation **collect** that performs the reduction operation on the elements of the stream using **Collectors**.
- **Predefined Collectors**
  - **java.util.stream.Collectors** class containing many factory methods that provides most commonly used Collector implementations.
- **Some of the functionality offered by Collectors.**
  - Collecting elements to a java.util.Collection
  - Joining String elements to a single String
  - Grouping elements by custom grouping key

Collectors (contd..)

- **Collecting as Collections**
  - Collecting stream elements to a **Collection** is the most widely used operation.
  - Collectors class provide various methods that returns a collector which will then collect stream elements to a specific collection container.
- **toList() Method**
  - Returns a Collector that will accumulate stream elements into ArrayList in the encountered order.
- **toSet() Method**
  - Returns a Collector that will accumulate stream elements into HashSet object.
- **toMap(Function<T, K> keyMapper, Function<T, U> valueMapper): Method**
  - Returns a Collector that accumulates elements into a Map whose keys are derived from keyMapper function and values are from valueMapper function.

Collectors (contd..)

- **Collecting in String**
  - Collectors utility class provides some of overloaded methods that concatenates stream elements into a single string either by separating them with a delimiter if provided.
- **Example**

```
List<Person> persons = new ArrayList<>();
String result= persons.stream().filter(person->person.getAge()>20)
.map(Person::getLastName).collect(Collectors.joining(","));
```

- Result is List of String with all the names of people in the persons, older than 20

Optional class

- **Optional**
  - It is class which is part of j**ava.util package**. It is used to represent a value is present or absent.
  - **java.util.Optional** class should be used to avoid **NullPointerException** in our code.
  - Optional class will check for null value and if null value is present then it will return empty Optional instance.
  - Like Collections and arrays, it is also a Container to hold at most one value.
- **Advantages of Optional Class**
  - Null checks are not required no more **NullPointerException** at run time. Thus supports us in developing clean and neat Java Applications.
  - Remove boiler plate code

Optional Class Methods

infogain
Engineering Business Outcomes

Method	Description
Optional.of()	Creates and returns the Optional instance for the given class.
Optional.empty()	It creates an empty Optional object.
isPresent()	It returns true if the given Optional object is non-empty. Otherwise it returns false.
get()	It returns a reference to the item contained in the optional object, if present, otherwise throws NoSuchElementException.

Optional Method(Contd..)

Method	Description
ifPresent(Consumer<T> consumer)	It takes Consumer instance as an argument and if value is present then it run the given consumer passed as an argument otherwise do nothing.
orElse(T other)	It returns the value, if present inside optional, otherwise returns the value passed inside orElse() method
Optional.filter()	filter() method takes Predicate instance as an argument. The value in Optional instance is filtered and if filtered value is not empty then value is returned otherwise empty Optional instance is returned.
Optional.map()	Map() method takes Function instance as an argument in the form of lambda expression runs the given lambda expression if the Optional instance is not null otherwise it returns empty Optional instance.

infogain
Engineering Business Outcomes

Contact :

Infogain Corporation, HQ

485 Alberto Way
Suite 100
Los Gatos, CA 95032

Phone: 408-355-6000

Irvine

41 Corporate Park
Suite 390
Irvine, CA 92606

Phone: 949-223-5100

Austin

111 W. Anderson Lane
Suite E336
Austin, TX 78752

Phone: 512-212-4070

Phoenix

21410 North 19th Ave
Suite 114
Phoenix, AZ 85027

Phone: 602-455-1860

London

Millbank Tower, Citibase 21-24
Millbank, office no 1.7
London SW1P 4DP

Phone: +44 (0)20 3355 7594

Noida

A-16 & 21, Sector 60
Gautam Budh Nagar
Noida – 201 301, India

Phone: +91 12 0244 5144

Mumbai

Unit No. 74, 2nd Floor, SDF 3
SEEPZ, Andheri East
Mumbai – 400 096, India

Phone: +91 22 6114 6969

Bengaluru

#7, 18th Main Road, 7th Block
Koramangala
Bengaluru - 560 095, India

Phone: +91 80 4110 4560

Dubai

Office No. 255, Building No. 17
Dubai Internet City
Dubai, UAE

Phone: +971-4-458-7336

Singapore

144 Robinson Road, #13-01
Robinson Square
Singapore 068908

Phone: +65 62741455